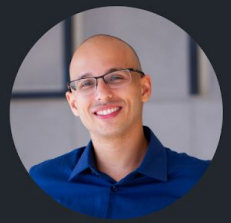
IMERSÃO .NET EXPERT

Treinamento C# Avançado



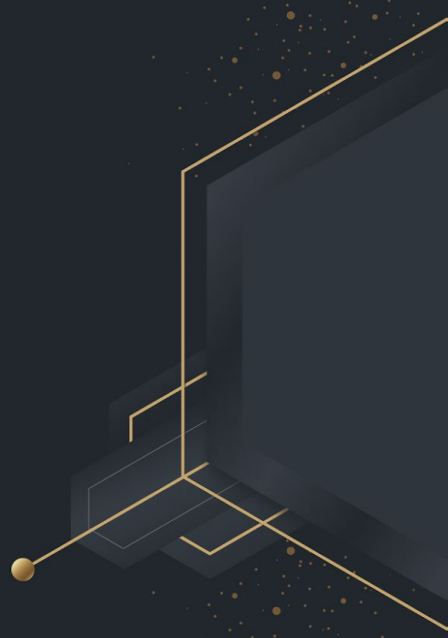
Luis Felipe

Desenvolvedor .NET Sênior,
3x Microsoft MVP

Cassiano Nunes

Arquiteto de Software

Sobre a Treinamento C# Avançado

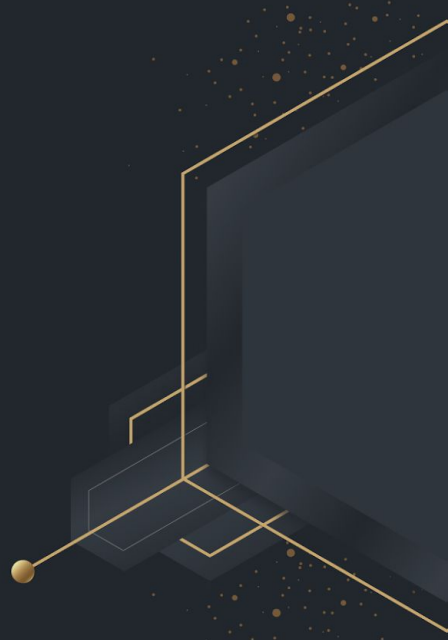


Sobre o Treinamento C# Avançado

O Treinamento C# Avançado é um treinamento para quem deseja se tornar referência técnica na linguagem, estando pronto para responder perguntas em entrevistas técnicas e aplicando conceitos em seus projetos

- **Instrutor**
    - Luis Felipe, Dev .NET Sênior, e Instrutor @LuisDev, 3x Microsoft MVP
- **Conteúdos**
    - Estruturas do C#
    - OOP avançada
    - Delegates e Events
    - Async-Await e Multithreading

Estruturas do C#



Estruturas de C#

- Além dos tipos mais básicos da linguagem, existem alguns que em maior ou menos grau vamos vir a utilizar, mas que exigem maior compreensão de suas características
- Entender isso é primordial para nos destacarmos como profissionais, afinal, o conhecimento nos ajuda tanto em nossos projetos quando em entrevistas
- As estruturas tratadas neste treinamento são
    - Enumerações
    - Struct
    - Record
    - Tuplas

Enumerações

Enumerações

- Permite a definição de constantes nomeadas, fortemente tipadas
- Entre os seus principais benefícios estão
  - **Legibilidade de código:** permite a definição de nomes claros para valores constantes, tornando o seu código mais legível e auto-explicativo, ao contrário do uso de strings ou números mágicos
  - **Segurança em Tipagem:** garante que apenas valores válidos de enums são atribuídos, permitindo detectar erros em tempo de compilação ao invés de execução
  - **Suporte ao Intellisense:** facilidade de utilização do enum com a ajuda do Intellisense
  - **Facilidade de Refatoração:** permite atualização do valor em apenas um lugar e será atualizado em todos seus usos

Enumerações

- Permite a definição de constantes nomeadas, fortemente tipadas
- Entre as suas principais limitações estão
    - **Formatação em String:** necessidade de utilizar atributos para definir formatação de um valor de enum para string
    - **Limitação de valores:** como são baseados em inteiros, tem uma quantidade limitada de valores únicos a serem utilizados
    - **Compatibilidade com bancos de dados:** apesar de que algumas ferramentas ORM resolvam esse problema, algumas exigem configuração adicional. Além disso, eles podem não mapear de forma correta caso não utilize uma ORM, dependendo do uso.

Struct

Struct

- Structs, ao contrário de classes, são de tipo de valor em C#.
- Entre os seus principais benefícios estão
    - **Performance:** eles são armazenados na stack, e tem uma sobrecarga menor na memória quando comparados com classes, que são alocadas na Heap, resultando em melhor performance em cenários onde temos várias instâncias criadas e destruídas rapidamente
    - **Não Nulas:** elas não podem ser nulas, então evitamos problemas de referência nulo que podem ocorrer com classes
    - **Semântica de Valor:** struct são copiados por valor, evitando problemas causados por alterações de instâncias/referências

Struct

- Structs, ao contrário de classes, são de tipo de valor em C#.
- Entre as suas principais limitações estão
    - **Sobrecarga na Performance:** apesar de structs oferecerem benefícios de performance em certos cenários, eles podem ter problemas de performance, como:
        - **Comportamento de Cópia:** como eles são copiados por valor, caso você passe um struct como parâmetro ou atribua a uma nova variável será feita uma cópia completa dele. Nesses cenários, pode apresentar problemas de formance, especialmente para structs grandes
            - A Microsoft indica um struct a ter no máximo 16 bytes

Struct

- Structs, ao contrário de classes, são de tipo de valor em C#.
- Entre as suas principais limitações estão
    - **Sobrecarga na Memória:** em alguns cenários, a sobrecarga na memória pode superar os benefícios do armazenamento na stack
        - **Alocação na Stack vs Heap:** a stack tem um tamanho limitado, logo, se o struct for muito grande, poderá resultar em problemas de overflow
            - A cópia de struct grandes resultam na cópia de muitos campos, aumentando o consumo de memória

Record

Record

- Records são um tipo de referência, que oferece uma maneira simplificada de criar classes imutáveis.
- Entre os seus principais benefícios estão
    - **Sintaxe Concisa:** introduzem uma sintaxe mais concisa para a criação de classes de dados imutáveis. É possível definir uma record com menos código, reduzindo a verbosidade.
    - **Semântica Imutável Padrão:** elas são imutáveis por padrão. Isso significa que seus valores não podem ser modificados após a criação, evitando erros comuns relacionados a mutabilidade.
    - **Métodos de Igualdade e Comparação Automatizados:** As records geram automaticamente os métodos Equals, GetHashCode e ToString, o que simplifica a comparação e a utilização em coleções, sem a necessidade de implementação manual.

Record

- Records são um tipo de referência de valor, que oferece uma maneira simplificada de criar classes imutáveis.
- Entre as suas principais limitações estão
  - **Desempenho:** usar records para todas as classes de dados pode não ser apropriado em todas as situações. Para classes de dados pequenas e frequentemente mutáveis, a criação de records pode ser excessiva.
  - **Compatibilidade com Padrões Existentes:** introduzir records em um código existente pode exigir ajustes em partes do código que lidam com igualdade e comparação personalizada.
  - **Decisão de uso:** records são imutáveis por padrão, o que pode ser um desafio se você precisar de uma classe mutável. Você terá que recorrer ao uso de classes normais em tais casos, decidindo caso a caso

Tuplas

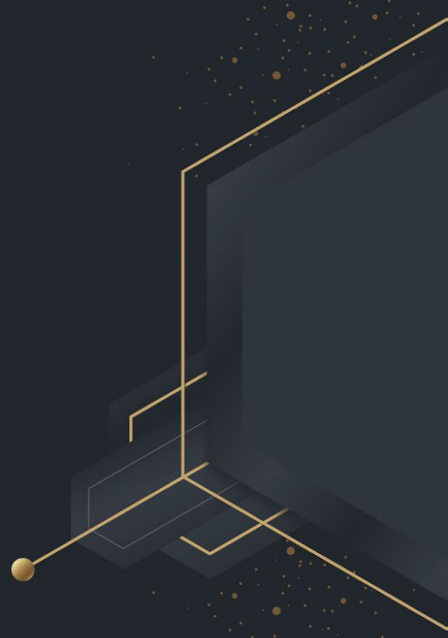
Tuplas

- Tuplas são uma estrutura de dados que permite agrupar um conjunto ordenado e heterogêneo de elementos em uma única unidade.
- Entre os seus principais benefícios estão
  - **Praticidade:** são convenientes para agrupar valores relacionados sem a necessidade de criar estruturas complexas.
  - **Performance:** em muitos casos, as tuplas podem ser mais eficientes em termos de memória e desempenho do que criar classes ou mesmo structs.
  - **Código mais limpo:** ajudam a simplificar o código ao eliminar a necessidade de criar classes ou structs simples apenas para agrupar valores.
  - **Retorno Múltiplo:** permitem retornar vários valores de um método ou função sem a necessidade de criar tipos personalizados.

Tuplas

- Tuplas são uma estrutura de dados que permite agrupar um conjunto ordenado e heterogêneo de elementos em uma única unidade.
- A sua principal limitação é
  - **Semântica limitada:** as tuplas não têm semântica rica como classes ou structs, o que pode dificultar a compreensão do código em cenários mais complexos. Se não tiver retorno nomeado, torna o código de difícil manutenção.
    - É importante entender a questão a ordem de elementos, principalmente na desestruturação de Tuplas

OOP Avançado



Herança e Polimorfismo

Herança e Polimorfismo

- A herança e o polimorfismo são conceitos fundamentais na programação orientada a objetos, incluindo a linguagem C#.
- Apesar de serem considerados básicos, são conceitos que quando compreendidos a fundo, permitem o seu uso em níveis altos de programação
- Esses conceitos permitem criar hierarquias de classes e a capacidade de tratar objetos de diferentes classes de maneira uniforme.

Herança e Polimorfismo

- Os principais benefícios de Herança
  - **Reutilização de código:** permite que classes herdem membros e métodos de outras classes, evitando a duplicação de código e promovendo a reutilização
  - Hierarquia de classes: permite criar uma hierarquia de classes, onde classes mais específicas podem herdar comportamentos e propriedades de classes mais gerais. Isso ajuda a modelar o domínio de maneira mais eficaz.
  - Polimorfismo: é a base para o polimorfismo, que permite que objetos de classes derivadas sejam tratados como objetos da classe base, facilitando o desenvolvimento de código genérico e flexível.

Herança e Polimorfismo

- Os principais desafios de Herança
  - **Acoplamento Forte:** pode levar a um acoplamento forte entre classes, o que pode dificultar a manutenção e a evolução do código. Mudanças na classe base podem afetar as classes derivadas.
  - **Hierarquias Complexas:** à medida que a hierarquia de classes cresce, pode se tornar complexo entender e manter a relação entre as classes. Isso pode levar a estruturas de herança confusas e difíceis de gerenciar.
  - **Herança Múltipla Limitada:** C# não suporta herança múltipla de classes, o que pode ser um desafio quando há a necessidade de herdar comportamentos de várias classes.

Herança e Polimorfismo

- Os principais benefícios de Polimorfismo
  - **Flexibilidade:** permite que um objeto de uma classe derivada seja tratado como um objeto da classe base, proporcionando flexibilidade na manipulação de objetos diferentes de maneira uniforme.
    - Alguns padrões de projeto são fortemente associados com esse benefício, como Strategy e Factory Method
  - **Código Genérico:** permite escrever código genérico que pode trabalhar com objetos de várias classes derivadas, o que aumenta a eficiência e a reutilização.
  - **Extensibilidade:** O polimorfismo facilita a extensão do comportamento das classes derivadas sem afetar o código existente que lida com objetos da classe base

Herança e Polimorfismo

- Os principais desafios de Polimorfismo
  - **Complexidade:** pode tornar o código mais complexo, pois os detalhes específicos das classes derivadas podem não ser evidentes no código que manipula objetos da classe base
  - **Desempenho:** em alguns casos, a resolução dinâmica de métodos associados ao polimorfismo pode introduzir uma sobrecarga de desempenho em comparação com chamadas de métodos diretos
  - **Entendimento:** em situações complexas, pode ser desafiador entender como um método específico será executado e qual será o comportamento real

Composição

Composição

- A composição envolve criar classes que contêm objetos de outras classes como membros internos. Em vez de herdar comportamento, uma classe se relaciona com outras classes através de composição, delegando responsabilidades para os objetos que contém.
- Os principais benefícios da Composição
  - Flexibilidade: permite combinar diferentes objetos para criar novos comportamentos sem herança direta.
  - Baixo Acoplamento: as classes são menos acopladas, facilitando a substituição de componentes e a manutenção.
  - Módulos Reutilizáveis: Os objetos podem ser criados e reutilizados independentemente em várias partes do código.

Composição

- Os principais desafios de Composição
  - **Gerenciamento:** é necessário gerenciar a criação, destruição e vida útil dos objetos compostos.
  - **Maior Complexidade:** em algumas situações, a composição pode levar a um código mais complexo, com muitos objetos interdependentes**.**
    - Sabe aquela situação onde uma classe tem vários objetos sendo recebidos via injeção de dependência? Estamos falando disso!

Interfaces e Classes Abstratas

Interfaces e Classes Abstratas

- As interfaces definem contratos que as classes que a implementam devem seguir, especificando métodos e propriedades que devem ser fornecidos
- Entre os seus principais benefícios estão
  - **Contratos Claros:** definem contratos bem definidos que as classes devem cumprir, promovendo a consistência no código
  - **Múltiplas Implementações:** uma classe pode implementar várias interfaces, permitindo que ela cumpra múltiplos contratos
  - **Desacoplamento:** promovem baixo acoplamento, permitindo que as classes se comuniquem sem conhecer os detalhes internos umas das outras.
  - **Testabilidade:** facilitam a criação de testes de unidade, pois as dependências podem ser substituídas por implementações de interface mock

Interfaces e Classes Abstratas

- As interfaces definem contratos que as classes que a implementam devem seguir, especificando métodos e propriedades que devem ser fornecidos
- Entre as suas principais limitações estão
  - **Código Repetitivo:** implementar uma interface em várias classes pode levar a código repetitivo se os métodos são semelhantes, levando a uma análise se vale a pena utilizar uma classe abstrata nesse cenários
  - **Mudanças na Interface:** mudanças podem afetar várias classes que a implementam, exigindo ajustes em várias partes do código
  - **Complexidade Adicional:** em projetos pequenos e com tempo de vida curto, interfaces podem adicionar complexidade desnecessária.

Interfaces e Classes Abstratas

- As classes abstratas fornecem uma base para outras classes, mas não podem ser instanciadas diretamente. Elas podem conter métodos abstratos (sem implementação) que devem ser implementados pelas classes derivadas
- Entre os seus principais benefícios estão
    - **Compartilhamento de Código:** podem fornecer implementações comuns de métodos que podem ser compartilhados por várias subclasses
    - **Definição de Estrutura:** podem definir uma estrutura básica para classes derivadas seguirem
    - **Métodos Abstratos:** podem conter métodos abstratos que as subclasses devem implementar, garantindo a conformidade com a estrutura da classe base

Interfaces e Classes Abstratas

- As classes abstratas fornecem uma base para outras classes, mas não podem ser instanciadas diretamente. Elas podem conter métodos abstratos (sem implementação) que devem ser implementados pelas classes derivadas
- Entre as suas principais limitações estão
    - **Restrições de Herança:** as subclasses só podem herdar de uma classe abstrata, limitando a flexibilidade em comparação com interfaces
    - **Acoplamento com a Hierarquia de Classes:** modificar a classe abstrata pode afetar todas as subclasses, o que pode levar a um alto grau de acoplamento

Generics

Generics

- Os generics são um recurso do C# que permitem criar classes, interfaces e métodos que funcionam com tipos de dados específicos, sem comprometer a segurança de tipos
- Entre os seus principais benefícios estão
    - **Reutilização de Código:** permitem criar código reutilizável que funciona com diversos tipos de dados, eliminando a necessidade de duplicar código para diferentes tipos
    - **Segurança na Tipagem:** mantêm a segurança de tipos em tempo de compilação, evitando erros de tipo em tempo de execução e aumentando a confiabilidade do código
    - **Performance:** podem melhorar o desempenho, pois evitam a necessidade de conversões de tipo desnecessárias

Generics

- Os genéricos são um recurso do C# que permitem criar classes, interfaces e métodos que funcionam com tipos de dados específicos, sem comprometer a segurança de tipos
- Entre os seus principais benefícios estão
    - **Flexibilidade:** classes genéricas podem ser usadas com uma ampla variedade de tipos de dados, proporcionando uma maior flexibilidade na criação de estruturas de dados e algoritmos

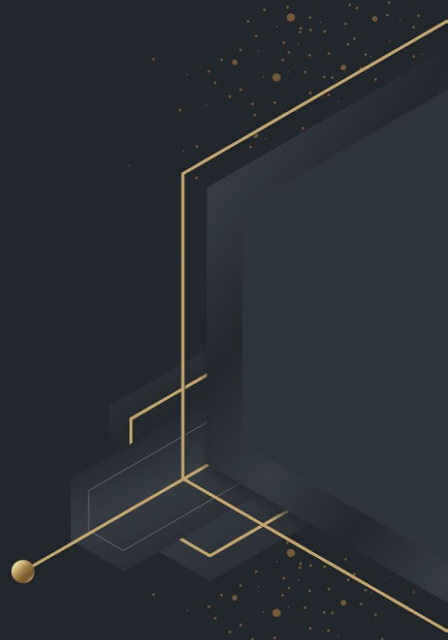
Generics

- Os genéricos são um recurso do C# que permitem criar classes, interfaces e métodos que funcionam com tipos de dados específicos, sem comprometer a segurança de tipos
- Entre as suas principais limitações estão
    - **Complexidade:** o uso incorreto de genéricos pode levar a código mais complexo e difícil de entender, especialmente quando há uma variedade de tipos envolvidos
        - Infelizmente é comum encontrar projetos que buscam a todo custo generalizar tudo, mesmo em cenários não compatíveis. Isso resulta em código de difícil manutenção, cheio de condicionais
        - Eu chamo isso carinhosamente de "Código Megazord"

Generics

- Os genéricos são um recurso do C# que permitem criar classes, interfaces e métodos que funcionam com tipos de dados específicos, sem comprometer a segurança de tipos
- Entre as suas principais limitações estão
    - **Restrições:** algumas operações podem ser restritas por limitações de tipo ou restrições genéricas, o que pode exigir soluções alternativas ou abordagens diferentes
    - **Curva de Aprendizado:** entender e aplicar corretamente generics pode exigir um certo nível de conhecimento e experiência, especialmente ao lidar com cenários avançados

Delegates e Events



Delegates

Delegates

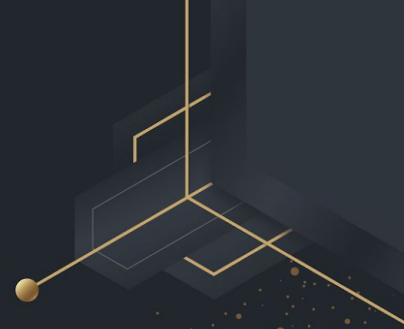
- Delegates são tipos de referência que permitem que você encapsule e passe métodos como parâmetros para outros métodos. Um delegate possui uma assinatura que corresponde à assinatura do método que ele pode referenciar, permitindo que você chame diferentes métodos usando o mesmo delegate
- Entre os seus principais benefícios estão
    - **Reutilização de Código:** permitem a reutilização de código, pois você pode passar o mesmo delegate para vários métodos, evitando a duplicação de lógica
    - **Flexibilidade:** proporcionam flexibilidade ao permitir que diferentes métodos sejam vinculados a um mesmo delegate. Isso permite a substituição de implementações sem alterar o código existente

Delegates

- Delegates são tipos de referência que permitem que você encapsule e passe métodos como parâmetros para outros métodos. Um delegate possui uma assinatura que corresponde à assinatura do método que ele pode referenciar, permitindo que você chame diferentes métodos usando o mesmo delegate
- Entre os seus principais benefícios estão
    - **Eventos Personalizados:** são a base para a criação de eventos personalizados, permitindo que objetos comuniquem mudanças e interações entre si
    - **Separação de Responsabilidades:** usar delegates para callback separa a lógica de chamada de métodos da lógica de implementação

Delegates

- Delegates são tipos de referência que permitem que você encapsule e passe métodos como parâmetros para outros métodos. Um delegate possui uma assinatura que corresponde à assinatura do método que ele pode referenciar, permitindo que você chame diferentes métodos usando o mesmo delegate
- Entre os seus principais desafios estão
    - **Dificuldade de Entendimento:** podem tornar o código mais complexo, especialmente quando usados de forma extensiva, o que pode dificultar a compreensão para desenvolvedores menos experientes
    - **Depuração:** a depuração de problemas relacionados a delegates pode ser desafiadora, uma vez que eles podem apontar para métodos diferentes em diferentes momentos da execução
    - **Segurança:** podem permitir a chamada de métodos não autorizados, caso não sejam usados com cuidado



Func, Action, e Predicate

Func, Action, e Predicate

- São delegates pré-definidos em C#
  - Func
    - Define uma assinatura de método que retorne um valor
    - Por exemplo, o método Where utiliza um Func<T, bool>
  - Action
    - Define uma assinatura que não retorna um valor
    - Por exemplo, o método ForEach utiliza um Action<T>
  - Predicate
    - Define uma assinatura que retorna um valor booleano
    - Por exemplo, o FindAll utiliza um Predicate<T>

Events

Events

- Eventos são mecanismos que permitem que objetos comuniquem entre si por meio de notificações quando um evento ocorre.
- Eles são amplamente utilizados na programação orientada a eventos, onde um objeto, conhecido como publicador (publisher), notifica outros objetos, conhecidos como assinantes (subscribers) ou ouvintes (listeners), sobre mudanças ou ocorrências relevantes.
- Isso permite a criação de sistemas interativos, como interfaces de usuário que respondem a cliques de botão
- Além disso, bibliotecas famosas utilizem eles, como por exemplo o RabbitMQ.Client

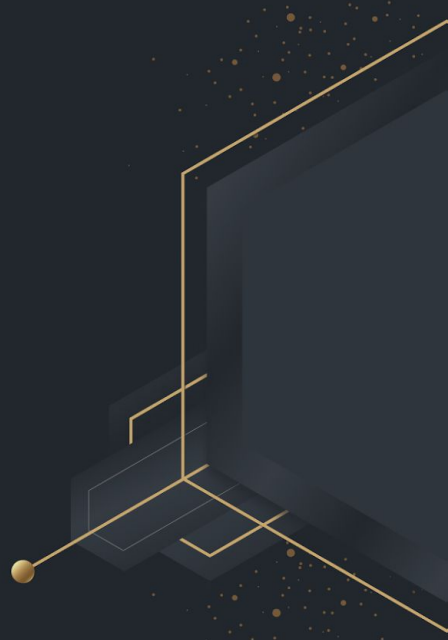
Events

- Entre os seus principais benefícios estão
  - **Desacoplamento:** promovem o desacoplamento entre componentes, pois os objetos que geram eventos não precisam conhecer os detalhes dos assinantes
  - **Flexibilidade:** assinantes podem ser facilmente adicionados ou removidos sem afetar o código do publicador
  - **Comunicação Eficiente:** permitem a comunicação eficiente entre diferentes partes de um aplicativo, facilitando a interação entre objetos
  - **Centralização:** permitem centralizar a lógica de notificação em um único local, tornando o código mais organizado
  - **Extensibilidade:** são fundamentais para a criação de bibliotecas reutilizáveis que podem ser estendidas por meio de assinantes

Events

- Entre os seus principais desafios estão
  - **Gerenciamento de Assinantes:** garantir que assinantes sejam corretamente adicionados e removidos para evitar vazamentos de memória pode ser desafiador
  - **Ordem de Execução:** a ordem em que os assinantes são notificados pode afetar o comportamento do aplicativo, e isso deve ser considerado ao projetar eventos
  - **Depuração:** identificar problemas em cenários de eventos pode ser mais complexo devido à natureza assíncrona e distribuída das notificações
  - **Complexidade:** podem adicionar complexidade ao desenho do código, especialmente em aplicações com muitos objetos interconectados por eventos

Async-Await e Multithreading



Daniel Abreu Dantas - contatodanieldantasdev@gmail.com - IP: 172.226.128.45

Async-Await

Async-Await

- Task vs. Thread
  - Em C#, uma tarefa (Task) é uma unidade de trabalho que pode ser executada em paralelo, mas não necessariamente em uma nova thread. Ela é gerenciada pelo runtime e pode ser agendada para execução em um thread do pool de threads
  - Uma thread, por outro lado, é uma unidade de execução de código. Threads são recursos mais pesados e custosos em termos de criação e gerenciamento do que Tasks

Async-Await

- A programação assíncrona é um paradigma de desenvolvimento de software que visa aprimorar a eficiência e a capacidade de resposta de um programa, especialmente quando se lida com operações que podem ser demoradas, como entrada/saída (I/O), incluindo acesso a bancos de dados, chamadas de rede, leitura de arquivos, e processamento de tarefas intensivas
- O conceito de programação assíncrona gira em torno da ideia de executar tarefas de forma não sequencial, permitindo que o programa continue avançando enquanto aguarda a conclusão de operações bloqueantes, como a leitura de um arquivo ou o envio de uma solicitação de rede
- Em .NET, isso é alcançado principalmente usando duas palavras-chave em C#: async e await

Async-Await

- **Async:** A palavra-chave async é usada para marcar um método como assíncrono. Isso indica ao compilador que o método pode conter operações assíncronas e que ele não será executado de maneira sequencial (bloqueante)
- **Await:** A palavra-chave await é usada para pausar a execução de um método assíncrono até que a operação assíncrona seja concluída. Enquanto aguarda a conclusão, o thread atual não fica bloqueado, permitindo que ele seja usado para outras tarefas
    - Ao não utilizar async-await em operações de I/O, existe o risco de um problema chamado Thread Starvation, onde muitas Threads ficam aguardando a execução
    - Suponha que várias chamadas HTTP ou de Banco de Dados, com longos tempos de espera, sem o uso de async await

Task Continuations

Task Continuations

- As Tarefas de Continuação são uma parte fundamental da programação assíncrona em C#
- Permitem que você defina o código que deve ser executado quando uma tarefa assíncrona é concluída. Isso oferece maior controle sobre o fluxo de execução e permite a composição de várias operações assíncronas em sequências lógicas

Task Continuations

- Entre os seus principais benefícios estão
  - **Sequenciamento de Operações Assíncronas:** as tarefas de continuação permitem que você encadeie operações assíncronas em uma sequência lógica. Isso simplifica a escrita de código assíncrono, tornando-o mais legível e compreensível
  - **Gerenciamento de Erros Simplificado:** ao usar tarefas de continuação, você pode lidar com exceções de maneira centralizada. Isso facilita o tratamento de erros em várias etapas de uma operação assíncron**a**
  - **Composição de Operações Assíncronas:** você pode criar um pipeline de operações assíncronas, onde cada tarefa de continuação realiza uma etapa específica após a conclusão da tarefa anterior.

Task Continuations

- Entre os seus principais desafios estão
  - **Complexidade de Leitura:** embora elas facilitem o encadeamento de operações, um grande número de tarefas de continuação pode tornar o código mais difícil de ler e entender, especialmente quando as operações são muito detalhadas
  - **Gerenciamento de Fluxo e Depuração:** se não forem configuradas adequadamente, você pode acabar com fluxos de execução complexos e difíceis de depurar
  - **Sincronização e Concorrência**: ao encadear várias tarefas de continuação, é importante considerar a sincronização e a concorrência, para evitar condições de corrida e resultados inesperados

Multithreading

Multithreading

- Multithreading refere-se à capacidade de um programa executar múltiplas threads (ou fluxos de execução) de forma concorrente em um único processo
- Cada thread representa uma sequência de instruções que é executada de forma independente das outras threads
- Em outras palavras, o multithreading permite que um programa execute várias tarefas simultaneamente, aproveitando os recursos de processamento disponíveis.

Multithreading

- Os benefícios principais relacionados a Multithreading
  - **Melhoria no Desempenho:** permite que um programa execute várias tarefas ao mesmo tempo, aproveitando ao máximo os recursos de hardware disponíveis e melhorando o desempenho geral
  - **Resposta Rápida:** Ao dividir tarefas em threads separadas, um programa pode responder a eventos externos, como entradas do usuário, enquanto continua a executar outras tarefas em segundo plano
  - **Utilização Eficiente do Processador:** permite que o processador seja usado de forma mais eficiente, especialmente em sistemas com vários núcleos de CPU

Multithreading

- Os desafios principais relacionados a Multithreading
  - **Condições de Corrida:** quando várias threads acessam recursos compartilhados simultaneamente, podem ocorrer condições de corrida, levando a resultados imprevisíveis e erros
  - **Sincronização e Bloqueio:** a sincronização adequada entre threads requer cuidado para evitar bloqueios excessivos ou subutilização de recurso. A sincronização refere-se à coordenação entre threads para garantir que o acesso a recursos compartilhados seja feito de forma segura e consistente, evitando condições de corrida e problemas de concorrência
  - **Deadlocks:** deadlocks ocorrem quando duas ou mais threads ficam bloqueadas, esperando umas pelas outras para liberar recursos
  - **Debugging Complexo:** depurar problemas de concorrência pode ser complexo, pois os erros podem ser intermitentes e difíceis de reproduzir.

Semaphore e Locks

Semaphore e Locks

- Locks e Semaphores são mecanismos de sincronização usados para controlar o acesso concorrente a recursos compartilhados em ambientes multithread
- Eles são especialmente importantes em aplicações onde vários threads competem pelo acesso a seções críticas do código
- O conceito de "lock" (ou bloqueio) refere-se a um mecanismo de sincronização usado em programação concorrente para garantir que apenas uma thread tenha acesso exclusivo a um recurso compartilhado por vez
    - O objetivo principal do uso de locks é evitar condições de corrida, onde várias threads tentam acessar ou modificar o mesmo recurso simultaneamente, levando a resultados indesejados ou inconsistentes.

Semaphore e Locks

- Quando uma thread deseja acessar o recurso compartilhado, ela solicita um lock no recurso
  - Se o lock estiver disponível, a thread obtém o acesso exclusivo ao recurso e o lock é adquirido.
  - Se o lock já estiver sendo mantido por outra thread, a thread solicitante fica bloqueada até que o lock seja liberado
- Benefícios relacionados ao Lock
  - **Sincronização:** garante que apenas uma thread possa acessar o recurso compartilhado por vez, prevenindo condições de corrida e mantendo a consistência dos dados
  - **Evita Conflitos:** evita conflitos quando várias threads tentam modificar o mesmo recurso simultaneamente

Semaphore e Locks

- Desafios relacionados ao Lock
  - **Deadlocks:** um deadlock pode ocorrer se duas ou mais threads ficarem bloqueadas, cada uma esperando que a outra libere um recurso
  - **Overhead de Desempenho:** o uso indiscriminado de locks pode levar a um overhead de desempenho, pois pode resultar em bloqueios excessivos e atrasos

Semaphore e Locks

- O conceito de "semáforos" é um mecanismo mais abrangente de sincronização usado para controlar o acesso a um número específico de recursos compartilhados
- Um semáforo age como um controlador que gerencia a quantidade de acesso permitido a um determinado recurso ou seção crítica
- Possui um valor inteiro que pode ser incrementado ou decrementado pelas threads. Quando o valor do semáforo é maior que zero, as threads podem adquirir acesso aos recursos
- Quando o valor do semáforo é zero, as threads ficam bloqueadas até que o semáforo seja incrementado por outra thread.

Semaphore e Locks

- Entre os principais benefícios relacionados a Semáforos
  - **Controle de Acesso:** permitem controlar o acesso a um número específico de recursos, evitando a sobrecarga de recursos compartilhados
  - **Gerenciamento de Recursos:** são úteis para gerenciar recursos limitados, como conexões de rede, pool de threads e semelhantes
- Entre os principais desafios relacionados a Semáforos estão
  - **Complexidade:** a gestão e coordenação de múltiplos semáforos podem ser complexas, especialmente em sistemas com muitos recursos compartilhados
  - **Deadlocks:** assim como com locks, o uso inadequado de semáforos pode levar a deadlocks quando as threads ficam bloqueadas esperando uns pelos outros.

Interlocked

- A classe Interlocked em C# fornece operações atômicas para trabalhar com tipos numéricos primitivos
- Essas operações garantem que as operações de leitura, modificação e gravação sejam executadas de forma atômica, sem a necessidade de bloqueios explícitos
- Isso é particularmente útil em cenários multithreaded, onde várias threads podem acessar e modificar uma variável compartilhada simultaneamente.

Coleções Concorrentes

Coleções Concorrentes

- As coleções concorrentes são estruturas de dados projetadas especificamente para suportar acesso concorrente seguro a partir de várias threads
- Elas são uma parte fundamental da programação concorrente e multithreading, permitindo que várias threads acessem e manipulem dados compartilhados de maneira eficiente e sem a necessidade de bloqueios manuais.
- Principais coleções concorrentes
    - ConcurrentBag
    - ConcurrentDictionary
    - ConcurrentQueue
    - ConcurrentStack

Coleções Concorrentes

- Principais coleções concorrentes
  - **ConcurrentBag:** uma coleção não ordenada que pode armazenar elementos em ordem aleatória. É útil para cenários em que a ordem dos elementos não é importante
  - **ConcurrentDictionary:** é uma coleção de chave-valor que permite o acesso simultâneo seguro por várias threads. É eficiente para cenários que requerem acesso rápido e seguro a elementos por meio de chaves
  - **ConcurrentQueue:** é uma fila concorrente que oferece suporte a operações de enfileiramento (enqueue) e desenfileiramento (dequeue) seguras por várias threads
  - **ConcurrentStack:** é uma pilha concorrente que oferece suporte a operações de empilhamento (push) e desempilhamento (pop) seguras por várias threads

Coleções Concorrentes

- Benefícios relacionados a coleções concorrentes
  - **Segurança de Threads:** garantem que as operações de leitura e escrita possam ocorrer simultaneamente sem resultar em condições de corrida ou resultados incorretos
  - **Desempenho Aprimorado:** Ao contrário de usar bloqueios manuais, as coleções concorrentes são otimizadas para permitir acesso simultâneo e paralelo, resultando em um melhor desempenho em cenários concorrentes
  - **Redução de Deadlocks:** como elas são projetadas para suportar acesso concorrente, a necessidade de bloqueios manuais é minimizada, o que reduz a probabilidade de ocorrer deadlocks

Coleções Concorrentes

- Desafios relacionados a coleções concorrentes
  - **Complexidade:** a utilização delas pode ser mais complexa do que simplesmente usar coleções tradicionais. Os desenvolvedores precisam entender os comportamentos específicos dessas coleções
  - **Comportamento Não Determinístico:** a ordem em que as operações concorrentes são executadas pode levar a resultados não determinísticos em algumas situações, especialmente em cenários complexos
  - **Overhead de Memória:** algumas coleções concorrentes podem exigir um certo overhead de memória para armazenar informações de controle adicional, o que pode impactar o consumo de memória

Concluindo